Policy conflict analysis for diffserv quality of service management

Policy Conflict Analysis for DiffServ Quality of Service Management

Marinos Charalambides, Paris Flegkas, George Pavlou, Javier Rubio-Loyola, *Member, IEEE*
Arosha K Bandara, Emil C Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, *Member, IEEE*

***Abstract*— Policy-based management provides the ability to (re-)configure differentiated services networks so that desired Quality of Service (QoS) goals are achieved. This requires implementing network provisioning decisions, performing admission control, and adapting bandwidth allocation to emerging traffic demands. A policy-based approach facilitates flexibility and adaptability as policies can be dynamically changed without modifying the underlying implementation. However, inconsistencies may arise in the policy specification. In this paper we provide a comprehensive set of QoS policies for managing Differentiated Services (DiffServ) networks, and classify the possible conflicts that can arise between them. We demonstrate the use of Event Calculus and formal reasoning for the analysis of both static and dynamic conflicts in a semi-automated fashion. In addition, we present a conflict analysis tool that provides network administrators with a user-friendly environment for determining and resolving potential inconsistencies. The tool has been extensively tested with large numbers of policies over a range of conflict types.**

***Index Terms*— QoS management policies, Conflict detection, Dynamic conflict resolution**

I. Introduction

In recent years, policy-based management has been proposed as a suitable means for managing different aspects of IP networks, including Quality of Service (QoS) and security. Yet despite various research projects, standardization efforts and substantial interest from industry, policy-based management is still not a reality. There are some vendor tools, mostly for virtual private network provisioning, but policy-based management is still far from being widely adopted despite its potential benefits of flexibility and "constrained programmability". One of the reasons behind the reticence to adopt this technology is that it is difficult to analyze policies in order to guarantee configuration stability given that policies may have conflicts leading to unpredictable effects.

Although there has been considerable work on analysis of security policies, analysis of management policy has received comparatively little attention. Initial work in [1] identified policy conflicts in policy-based management as being analogous to software bugs. Subsequent work [2] focused on conflicts related to generic management policies and described means to statically detect conflicts, but did not take into account policy constraints that restrict the applicability of the policies involved. A logic-based approach was therefore introduced in [3], which provides advanced reasoning capabilities to cope with the emerging requirements of complex systems.

This paper generalizes our initial approach [5, 6], provides a comprehensive review of policy conflicts in QoS management for DiffServ networks and describes an integrated framework for policy analysis, conflict detection and resolution. The approach we adopt is based on the work presented in [3] where Event Calculus (EC) was proposed as the underlying formal representation for policies, systems behavior and for the rules that define the conditions that will result in conflicts and are therefore used to detect the presence of conflicts. In this approach, Ponder policies [7] and design-level models of managed objects such as state-charts are automatically translated to the EC representation to facilitate analysis and detect the presence of inconsistencies.

In contrast to prior work that only used examples from QoS management to illustrate specific techniques, we aim here to comprehensively cover QoS provisioning policies from service management to traffic engineering, and classify inconsistencies that may arise between them based on their properties. We re-visit static conflict detection in the context of service subscription policies and enhance our approach towards automating dynamic conflict analysis which can trigger conflict resolving policies. Our approach is implemented in an integrated tool supporting both static and dynamic conflict analysis, which has been extensively tested for scalability using large numbers of policies.

In the next section we present background information regarding the QoS management framework we adopt and our conflict analysis approach. This is followed by a detailed description of the various policies that can be used to drive the functionality of individual QoS modules in Section III.

Manuscript received March 1, 2008. This work was supported by EPSRC (Grant Nos: GR/S79985/01 and GR/S79992/01).

Marinos Charalambides is with the Centre for Communication Systems Research, University of Surrey, Guildford, GU2 7XH, UK (phone: +44-1483-683641; fax: +44-1483-686011; e-mail: m.charalambides@surrey.ac.uk).

P.Flegkas, G.Pavlou, and J.R.Loyola are all with the Centre for Communications Systems Research, University of Surrey, Guildford, GU2 7XH, UK (e-mail: {p.flegkas, g.pavlou, j.r.loyola}@surrey.ac.uk).

E.C. Lupu, A. Russo, N.Dulay, and M. Sloman are all with the Department of Computing, Imperial College London, London, SW7 2AZ, UK (e-mail:{e.c.lupu,a.russo,n.dulay,m.sloman}@imperial.ac.uk).

A.K.Bandara is with the Department of Computing, The Open University, Milton Keynes, MK7 6AA, UK (e-mail: a.k.bandara@open.ac.uk).

Section IV presents a classification of the conflict types identified and the conditions under which these conflicts arise. Section V elaborates on our conflict analysis approach and describes the tool developed. Section VI presents and discusses experimental results and analysis examples. Section VII reviews the results of this paper in the context of the related work in this area and Section VIII presents our conclusions and future work.

II. Background

A. DiffServ QoS Management Framework

In order to provide a holistic approach for QoS management in DiffServ networks, a range of management operations need to be deployed from traffic engineering and admission control, to dynamic management of resources. Several frameworks have been proposed for this purpose that mainly stemmed from European collaborative research projects including TEQUILA [8], MESCAL [9], and ENTHRONE [10]. All frameworks propose the use of a general model depicted on Fig. 1, where the QoS management goals are realized by two distinct management blocks: *Service Management* and *Traffic Engineering* (TE). The former is responsible for agreeing the customers' or peer domain's QoS requirements in terms of Service Level Specifications (SLSs), while Traffic Engineering is responsible for fulfilling contracted SLSs by deriving the network configuration.

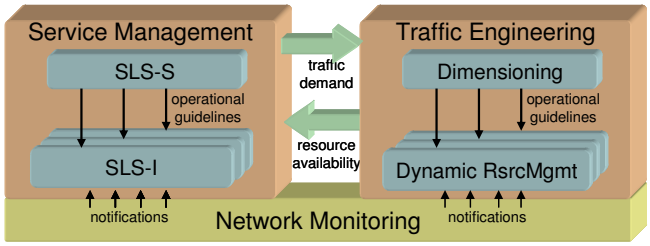

Fig. 1. QoS Management Framework. Two main sub-systems, Service Management and Traffic Engineering, each implementing a two-level module hierarchy.

The functionality of each of the two sub-systems is realized by a two-level hierarchy of modules that reflect the off-line and run-time operational mode of the model. On the service management side, the *SLS-Subscription (SLS-S)* module has a centralized off-line functionality and performs admission control on subscription requests based on resource availability provided by the TE system, whereas the *SLS-Invocation (SLS-I)* module is distributed across ingress routers and performs dynamic invocation of already subscribed SLSs based on the network state, following operational guidelines provided by the SLS-S. On the Traffic Engineering side, the *Network Dimensioning (ND)* module is a centralized off-line component that has a global view of the network. It maps forecasted traffic demands onto physical network resources in terms of MPLS Labeled Switched Paths (LSPs) and anticipated loading for each QoS class on all interfaces, providing a long to medium term network configuration. Configuration parameters are given in the form of an admissible range of values and are used by *Dynamic Resource Management (DRsM)* modules as guidelines for the run-time allocation of resources in response to short-term traffic variations.

The dynamic aspects of the architecture are supported by a monitoring sub-system that closes the management loop. It provides information about the current network state, e.g. link utilization, in the form of threshold-crossing alarms that trigger dynamic reconfiguration actions.

B. Approach to Policy Conflict Analysis

In an environment where a number of policies need to coexist, there is always the likelihood that several policies will be in conflict, either because of a specification error or because of application-specific constraints. It is therefore imperative to detect and resolve these conflicts.

Our approach is based on a *identify-classify-detect-resolve* principle, where an application expert would (a) analyze the various policy types governing the behavior of a managed system and identify possible inconsistencies that may arise among these policies, (b) classify the conflicts based on the nature of their occurrence, and, (c) develop techniques and mechanisms for their effective detection and resolution. Policy conflicts can be characterized by the policy types involved and the scope of their enforcement, application environment constraints, and the time frame at which they can be detected relative to policy enforcement. A classification based on such properties/characteristics is vital for the correct design of efficient detection rules as well as for the implementation of appropriate detection/resolution mechanisms.

Various conflict types have been identified in the literature, at different abstraction levels, spanning from generic policy conflicts in distributed systems management [1], [2] to more specific ones in the domain of QoS and security management [4], [35]. These were broadly classified into modality and application-specific conflicts, the former being conflicts that can be derived from the policy syntax such as positive/negative conflicts and the latter being specific to the application i.e., the subjects, targets and actions specified in the policy. Whilst for modality conflicts the conditions under which the conflicts arise are generic, for application specific conflicts these conditions need to be specified and encoded in rules that can detect the occurrence of a conflict. These rules are also sometimes called meta-policies, although the term is overloaded, and may include system-specific data in addition to policy information for correctly capturing conflicting situations. An instance of system-specific data here are the functional dependencies between the QoS management modules, which in effect define the scope of a conflict. These dependencies are implicit in the hierarchical QoS management framework described, where the functionality of the SLS-I and DRsM modules is constrained by guidelines provided from SLS-S and ND respectively. We can thus distinguish between *intra-* and *inter-module* conflicts, the former arising from policies applying to a single module, and the latter

arising between policies defined for different modules.

In addition to scope and level of abstraction, policy conflicts can also be classified based on the time-frame at which they can be detected. Thus we can distinguish between *static* conflicts that can be detected through off-line analysis at specification time and *dynamic* conflicts that can only be detected when policies are enforced and depend on the current state of the managed system [6]. These factors influence the analysis methodology and requirements for dealing with conflicts. Static conflicts are typically detected through analysis initiated manually by the system administrator; conflicts represent inconsistencies between policies and are typically resolved by amending the policies. In contrast, run-time conflicts must be detected by a process that monitors policy enforcement and detects inconsistent situations in the system's execution. Resolution must be achieved automatically, for example through enforcing resolution rules. Lack of automation in the handling of run-time conflicts may have catastrophic consequences on the correct system operation, especially when managing QoS for delay sensitive applications.

Our conflict analysis approach is based on formal methods and derivation that caters for the various conflict types we have identified in policy-driven service management and traffic engineering. Based on previous work [3] we have chosen to use *Event Calculus* for this purpose as it permits representation of events and persistent properties. The formalism is used for the representation of both policies and the managed system [11], providing a uniform description that is amenable to analysis. Analysis relies on specifying detection rules to define the conditions for a conflict. As discussed in Section V-A&B, resolution for the identified static conflicts largely requires human intervention, in contrast to dynamic conflicts for which resolution is automated and comes in the form of predefined policies. For demonstration purposes, the dynamic analysis process developed operates on Event Calculus-based models that emulate the behavior of on-line QoS management modules.

C. Formal Representation and Event Calculus

Event Calculus (EC) is a logic formalism for representing and reasoning about dynamic systems. Because it supports a time representation that is independent of any events that may occur, it provides a particularly useful way to specify a variety of event-driven systems. In the context of our work, EC serves as the underlying formalism for describing policies and the managed system since it has well understood semantics, and supports all modes of logical reasoning.

Since its initial presentation [12], a number of variations have been presented in the literature. In this work we use the form presented in [13], consisting of (i) a set of time points (that can be mapped to the non-negative integers); (ii) a set of properties that can vary over the lifetime of the system, called *fluents*; and (iii) a set of event types. In addition, the language includes a number of base predicates: `initiates`, `terminates`, `holdsAt`, `happens`, as summarized on Table I.

TABLE I
EVENT CALCULUS BASE PREDICATES

Predicate	Description
initiates(A, B, T)	Event A initiates fluent B for all time > T
terminates(A, B, T)	Event A terminates fluent B for all time > T
happens(A, T)	Event A happens at time point T
holdsAt(B, T)	Fluent B holds at time point T
initiallyTrue(B)	Fluent B is initially true
initiallyFalse(B)	Fluent B is initially false

This is the classical form of Event Calculus where theories are written using Horn clauses. The frame problem is solved by circumscription, which allows the completion of the predicates `initiates`, `terminates` and `happens`, leaving open the predicates `holdsAt`, `initiallyTrue` and `initiallyFalse`. This approach allows the representation of partial domain knowledge (e.g. the initial state of the system). Formulae derived from Event Calculus are in effect derived from the circumscription of the EC representation.

The analysis we wish to perform must determine the policies that hold given some preconditions and a sequence of events. This involves performing deductive reasoning over the Event Calculus based formalism, taking into account rules containing applicability constraints for the policies. If we ensure that the formal representation conforms to a restricted version of normal logic programs – known as stratified normal logic programs – it can be shown that the analysis reasoning task is P-complete [14]. A stratified normal logic program is a program whose rules can be ordered such that for any rule that has a negated literal in its body, there is rule before it with that literal in the head. This can be achieved by ensuring that the behavioral specification of managed objects does not involve self-transitions – a restriction which does not affect our ability to model the behavior of the presented QoS management framework.

III. POLICY DRIVEN QOS MANAGEMENT

QoS management has always been one of the most popular application domains of policies since ISPs can realize their objectives through flexible programmability with respect to offered services and treatment of customer traffic in their network. A small number of policies have been defined for some of the components of the QoS management framework described in the previous section [15]. Here, we provide a comprehensive set of QoS management policies and explain how their enforcement defines the behavior of the managed modules and associated IP DiffServ managed objects. The majority of these policies are generic enough to apply to other QoS and resource management frameworks, where functions for admission control and BW management are essential. They are categorized into service management and traffic engineering policies and follow the two-level hierarchy of Fig. 1.

A. *Service Management Policies*

The service management functionality is realized by the SLS-S and SLS-I modules that perform static and dynamic admission control respectively. The main objective of subscription logic is to control the number and type of service subscriptions, aiming to avoid overloading the network, whilst at the same time maximizing subscribed traffic. To achieve this objective, the SLS-S module employs managed objects that expose a set of methods defining its programmable functionality. These methods guide the evolution of the SLS-S module in its operation through a number of states represented in the state machine diagram of Fig. 2. The transitions between these states can thus be managed through policies. The SLS-S operation distinguishes two main processes – initialization and admission control – during which parameters essential for the modules operation are initialized, and the actual decision on the acceptance/rejection of new subscriptions is taken; the relevant policy actions are summarized on Table II.

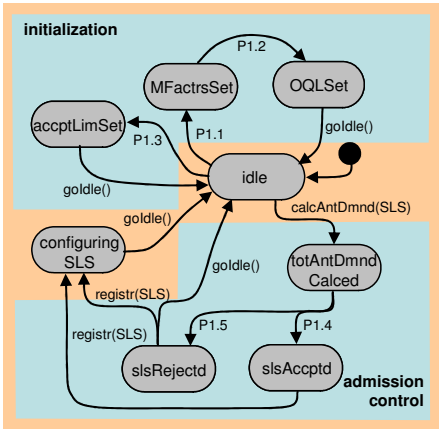

Fig. 2. SLS-S module behavior

TABLE II
SLS-S POLICY ACTIONS

ID	Policy action	Description
P1.1	setAlmstSatisf(QC, Fctr) setFullSatisf(QC, Fctr)	Sets the almost and full satisfaction factors per QC
P1.2	setQltLvl(QC, OQL)	Sets the overall quality level per QC
P1.3	setSUConsrv(Value) setSUModrt(Value) setSURisky(Value)	Sets the RAB upper limit in a conservative, moderate, or risky fashion
P1.4	accpt(SLS)	Accepts an SLS subscription request
P1.5	rejct(SLS)	Rejects an SLS subscription request

The first two actions, P1.1 and P1.2, use the notion of service satisfaction and quality levels [16] to set the relevant parameters per QoS Class (QC) and their values range from 0 to 1. Satisfaction parameters essentially define multiplexing factors that are used to derive the rates at which a service is considered *almost* and *fully* satisfied, and quality parameters are analogous to the confidence level with which a SLS is to enjoy the agreed QoS – quality values close to 1 being appropriate for high priority QCs. The example below encodes action P1.1 into policy specification following the Ponder format [7] and defines the almost satisfied factor for AF1 traffic during peak hours.

```
inst oblig /policies/sls-s/P1.1 {
        on    newRPC();
        subj  s = sls-sPMA;
        targ  t = sls-s/servSatisfMO;
        do    t.setAlmstSatisf(af1, 0.2);
        when  duration(08:00-18:00);
}
```

As mentioned in Section II-A, subscription admission control is based on resource availability and more specifically on a Resource Availability Buffer (RAB) [16] which holds aggregated traffic demand of subscribed SLSs on a per Traffic Trunk (TT) basis. Policy actions P1.3 set the upper limit – subscription upper (SU) – as a percentage of the RAB for accepting new subscriptions in a conservative, moderate, or risky fashion and define the level of associated risk in satisfying the QoS requirements. The acceptance limit is used as a constraint when deciding whether to accept or reject a request as in the policy example below encoding P1.4, where a request is accepted if the aggregated traffic demand is less than SU.

```
inst oblig /policies/sls-s/P1.4 {
        on    totalAnticipatedDemandCalced(SLS);
        subj  s = sls-sPMA;
        targ  t = sls-s/acMO;
        do    t.accept(SLS);
        when  t.getTotalDemand(SLS.tt) < t.getSU(SLS.tt);
}
```

In contrast to the static nature of the subscription module, the service invocation logic is based on run-time events/ notifications to regulate the traffic entering the network. The policies used here perform dynamic admission control on the number and types of active services, as well as on the volume and type of traffic admitted into the network. The behavior of the SLS-I module as a result of policies is depicted in Fig. 3 and the relevant actions supported are listed on Table III.

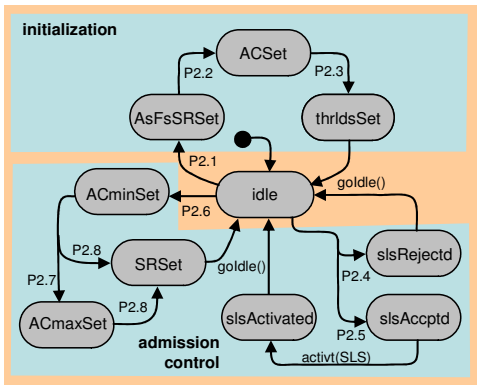

Fig. 3. SLS-I module behavior

The first three actions are invoked by policies during the initialization of the module and provide initial values to various parameters – P2.1 sets the rates that are thought to almost/fully satisfy a service, P2.2 sets minimum (AC_{min}) and maximum (AC_{max}) parameters of the admission control algorithm, and P2.3 defines two thresholds that signal target critical (TCL) and very critical levels (VCL) of traffic flowing into the network.

TABLE III
SLS-I POLICY ACTIONS

ID	Policy action	Description
P2.1	setSR$_{AS}$(TT, Value) setSR$_{FS}$(TT, Value)	Sets almost/fully satisfied service rates per TT
P2.2	setAC$_{min}$(TT, Value) setAC$_{max}$(TT, Value)	Sets admission control min/max limits wrt RAB per TT
P2.3	setTCL(TT, Value) setVCL(TT, Value)	Sets target/very critical level thresholds wrt RAB per TT
P2.4	rejct(SLS)	Reject an SLS invocation request
P2.5	accpt (SLS)	Accept an SLS invocation request
P2.6	incrAC$_{min}$(TT, Value) decrAC$_{min}$(TT, Value)	Increases/decreases admission control min parameter of a TT
P2.7	incrAC$_{max}$(TT, Value) decrAC$_{max}$(TT, Value)	Increases/decreases admission control max parameter of a TT
P2.8	incrSR(TT, Value) decrSR(TT, Value)	Increases/decreases the service rate of a TT

The run-time operation of the module is triggered by threshold crossing alarms and service invocation requests. The latter activate policy actions for accepting/rejecting a request (P2.4, P2.5), whereas the former initiate a set of actions that control the rates of incoming traffic (P2.8) and change invocation admission control parameters (P2.6, P2.7). These parameters are used as constraints in accept/reject policies and essentially define the treatment of new service invocations: the closer the aggregate value of current TT utilization and the requesting SLS traffic rate to AC_{max}, the less the chances of the SLS being successfully invoked. The example policies below encode actions P2.6 and P2.8 to handle the event of a TCL threshold crossing alarm. When enforced, they take proactive measures to avoid potential congestion built-up, by decreasing AC_{min} – thus decreasing the probability of accepting new invocations – and the service rate by 20% and 10% respectively.

```
inst oblig /policies/sls-i/P2.6 {
        on    TCLAlarmRaised(up, TT);
        subj  s = sls-iPMA;
        targ  t = sls-i/servAdjustMO;
        do    t.decrACmin(TT, 20);
        when  duration(08:00-18:00);
}
inst oblig /policies/sls-i/P2.8 {
        on    TCLAlarmRaised(up, TT);
        subj  s = sls-iPMA;
        targ  t = sls-i/servAdjustMO;
        do    t.decrSR(TT, 10);
        when  duration(08:00-18:00);
}
```

B. *Traffic Engineering Policies*

Traffic Engineering is responsible for fulfilling the contracted SLSs by deriving the long- and short-term network configuration. The former is realized by Network Dimensioning (ND) that maps the forecasted traffic demand provided by SLS-S onto physical network resources in terms of MPLS Labeled Switched Paths (LSPs) and anticipated loading for each DiffServ QoS class (QC) on all interfaces. The policy actions on Table IV guide the functional behavior of dimensioning to effectively achieve an optimal configuration in terms of network load.

TABLE IV
NETWORK DIMENSIONING POLICY ACTIONS

ID	Policy action	Description
P3.1	setNDMin(QC, BW) setNDMax(QC, BW)	Sets min/max allocation per QC
P3.2	setupLSP(QC, [Path], BW)	Sets explicit LSPs per QC through nodes of Path
P3.3	calcHopCountMin(QC) calcHopCountMax(QC) calcHopCountAvg(QC)	Derives the hop count constraint of ND algorithm per QC, with different strategies
P3.4	setMaxHops(QC, HopNum)	Sets the maximum number of hops per QC
P3.5	setMaxAltPaths(QC, [TT], PathNum)	Sets the maximum number of alternative paths per TT
P3.6	allocSpareBW( )	Distributes spare BW among QCs
P3.7	redOverBW( )	Reduces over-allocated BW

The first two policy actions, P3.1 and P3.2, allow an administrator to set upper and lower bounds of network capacity per QC, and setup explicit paths through which specific traffic will be routed. Another function of ND is to handle the QoS requirements of the expected traffic in terms of delay and packet loss. This process is simplified by transforming the delay and loss requirements into constraints for the maximum hop count for each traffic trunk. The actions P3.3 allow for different strategies in achieving this objective with *calcHopCountMax* being conservative and appropriate for high priority traffic. The core component of ND is an optimization algorithm and its objective is to find a set of paths for which the BW requirements of TTs are satisfied and the delay and loss requirements are met by using the hop count constraint as an upper bound. This is a non-linear optimization problem which is solved by the gradient projection method [25]. Action P3.4 sets an upper bound on the number of hops the calculated paths are permitted to have, and P3.5 defines the number of alternative paths the optimization algorithm should allow for every traffic trunk that belongs to a specific QC for the purpose of load balancing. In the final stages, ND assigns the residual physical capacity to the various traffic classes or reduces the allocated capacity because the link capacity is not enough to satisfy the predicted traffic requirements. Actions P3.6 and P3.7 can achieve these objectives with different strategies by explicitly, equally or proportionally reducing/distributing capacity between the various traffic classes.

TABLE V
DRSM POLICY ACTIONS

ID	Policy action	Description
P4.1	incrAlloc(Link, QC, BW)	Increases allocation per QC
P4.2	decrAlloc(Link, QC, BW)	Decreases allocation per QC
P4.3	incrThs(Link, QC, BW)	Increases tracking thresholds per QC
P4.4	decrThs(Link, QC, BW)	Decreases tracking thresholds per QC

The provisioning directives from ND are treated as nominal values and are a result of predicted demand. Dynamic TE functions are deployed by DRsM that deal with traffic fluctuations around the forecasted values in order to optimize resource utilization. The DRsM policy actions of Table V

manage available resources based on utilization monitoring.

Utilization monitoring generates events to trigger policy actions that increase/decrease resource allocation as well as utilization tracking thresholds using absolute (e.g. in kbps) or relative values (e.g. as a percentage). The example policy below encodes action P4.1 to increase the allocation for AF1 traffic by 10% on a particular link, in the event of an upper tracking threshold crossing alarm.

```
inst oblig /policies/drsm/P4.1 {
        on    drsmAlarmRaised(upprTh, link1, af1);
        subj  s = drsmPMA;
        targ  t = drsm/allocMO;
        do    t.incrAllocRel(link1, af1, 10);
        when  duration(08:00-18:00);
}
```

State chart behavior representations of both TE modules can be found in [5] and [6].

IV. QoS Management Policy Conflicts

Thus far we have described the behavior of the individual QoS management modules and provided example policies that would be used to manage these modules. In order to use these policies in a running system it is necessary to check that they do not conflict with policies already deployed in the system. In this section we present the different types of conflict that could arise between policies written for different modules in the QoS management system. We start by providing a taxonomy of the conflict types that have been identified.

A. Conflict Classification

We identified a number of potential conflicts related to policies that drive the QoS management modules' functionality, and classified them as shown in Fig. 4.

Although it would be possible to classify these conflicts using different characteristics we have chosen to distinguish the categories based on their level of abstraction, the subsystem in which they occur and their specificity to the application domain as we believe these most naturally reflect the scope in which they occur. First we distinguish between conflicts that are module-independent and those specific to the two management subsystems. Module-independent conflicts may occur among any of the QoS management policies, whilst service management and traffic engineering conflicts are specific to the operations supported by the relevant modules. The latter categories are further subdivided into conflicts relating to policies for individual QoS management modules (intra-module), and to policies applying to different modules (inter-module).

Module-independent conflicts represent the simplest forms of inconsistency that may arise between policy specifications and examples include *redundancy*, *mutual exclusivity* and *QC priority* conflicts. Redundancy conflicts may arise because of duplicate policies or policies with inconsistent action parameters in relation to others and can be detected by syntactic analysis. Mutual exclusion conflicts occur between policies implementing alternative strategies that realize the same goal. Examples of the latter conflict type include SLS-S policies for setting the upper limit in the RAB in a conservative, moderate, or risky fashion, ND policies defining the treatment of spare/over-provisioned BW, and DRsM policies managing the allocation on link resources through different strategies. The various actions are said to be mutually exclusive since there should not be more than one directive specifying an operation on a particular managed resource. An example of such inconsistency would be between a DRsM policy incrementing the resource allocation using an absolute value (e.g. 500 kbps) and policy P4.1 of the previous section. The conflict will materialize if the two policies are triggered by the same event, apply to the same link and QC, and have an overlap in the time constraints.

The relative priorities between traffic classes can cause inconsistencies to arise between policies defined on the various QCs in the context of any QoS management module. These are termed *qcPriority* conflicts and will materialize if the effect of a policy action violates the priority between QCs. Figure 4 shows two examples of such a conflict between SLS-S policies for setting service satisfaction and quality levels (P1.1 and P1.2). A *multiplex* conflict will occur if the multiplexing factor of a particular QC is higher than that of another QC with lower priority, whereas an *oql* conflict will arise if the quality level of a QC with high priority is lower than that of QC with lower priority.

The above module-independent conflicts, as well as some

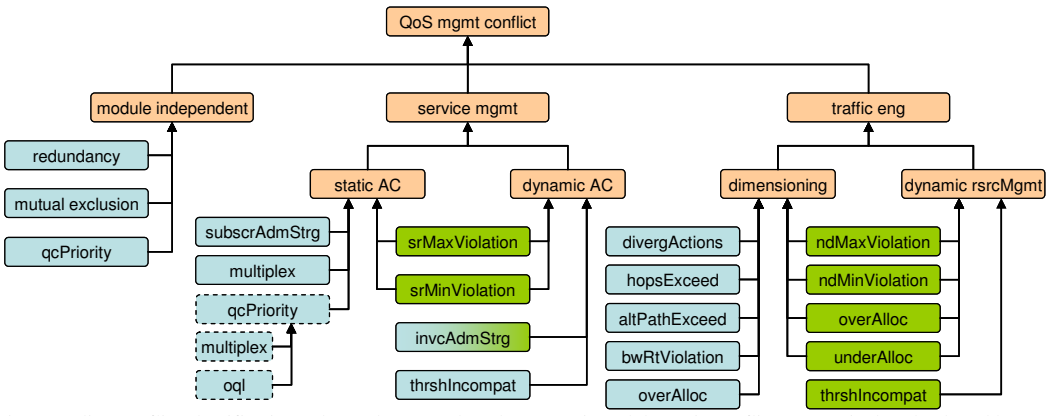

Fig. 4. Policy conflict classification. Blue and green colors denote static and dynamic conflicts respectively. The dotted box outline indicates instances of the module-independent *qcPriority* conflict pertaining to service management policies.

of the identified inconsistencies, can be determined through static analysis at policy-specification time. Policies governing the behavior of online modules, on the other hand, are mostly prone to conflicts that can only be detected dynamically at enforcement-time as their manifestation depends on the current state of the underlying managed resources. The next sub-sections describe the various conflicts relating to service management and traffic engineering policies and the conditions under which they arise.

B. Service Management Policy Conflicts

Conflicts specific to our application domain primarily occur because of inconsistent attribute values set by policies. It is essential that these are individually identified such that the exact reason for their occurrence can be defined and eventually resolved by a network administrator or in an automated manner.

Conflicts related to the SLS-S module can be detected at specification-time and arise between policies governing the process of static admission control. As mentioned previously, the upper limit in the RAB is a major factor for the decision of accepting/rejecting a new subscription request and can be defined with different strategies (P1.3): risky, moderate, and conservative. A *subscrAdmStrg* conflict will arise if the resulting value of the conservative approach is greater than that of moderate, or if the latter is greater than the one generated by the risky approach. *Multiplex* conflicts occur between service satisfaction policies (P1.1) that essentially define multiplexing factors used to derive the rates at which a service is considered *almost* and *fully* satisfied. This inconsistency will occur if the multiplexing factor for fully satisfied is greater than the multiplexing factor for the almost satisfied for the same QoS class, as they are inversely proportional to the service rates produced.

To regulate the traffic entering the network, SLS-I works on guidelines provided by SLS-S. These come in the form of policies that act as constraints and although harmonizing the operation of the two modules, they may cause dynamic conflicts that we term inter-module: *srMaxViolation* or *srMinViolation* conflicts occur when service invocation policy actions (P2.8) try to alter the service rate of a specific trunk but violate the almost or fully satisfied rate boundaries provided by the subscription policy P2.1. Besides activating service rate policies, threshold crossing alarms also trigger policies that manipulate AC parameters (P2.6, P2.7) aiming to provide proactive and reactive control over invoked services. The relative position of both thresholds and AC parameters in the RAB of each trunk allows the administrator to adapt the strategy by which services are admitted to the network and potentially avoid the build-up of congestion while maximizing resource utilization. Incorrect definition of these parameters will lead to an *invcAdmStrg* conflict that can occur both at policy specification-time, but also during system execution as AC parameters may be re-calculated on the fly. This inconsistency will arise if AC_{max} is less than VCL, or if AC_{min} is greater than TCL. Lastly, an intra-module static conflict – threshold incompatibility (*thrshIncompat*) – can occur between policies setting the threshold values (P2.3) if TCL – aiming to trigger some proactive measures – is greater than VCL.

C. Traffic Engineering Policy Conflicts

Conflicts between policies guiding the functional behavior of Network Dimensioning are static in nature and occur due to contradicting action parameters of BW allocation and routing policies. A *divergActions* conflict may arise between two policies setting the BW allocation boundaries for a specific QC (P3.1), if their parameter values do not converge. These policies may also cause an *overAlloc* conflict if the sum of the allocation corresponding to all the supported QCs exceeds 100%. The same rule applies to explicit actions responsible for the distribution of spare resources or the treatment of over-provisioned BW (P3.6, P3.7). *HopsExceed* conflicts occur if the hop count of the path, through which an LSP is set (P3.2), exceeds the maximum number of allowed hops specified in P3.5, provided both policies apply to the same QC. LSP policies can lead to further inconsistencies – *altPathExceed* and *bwRtViolation* – if their instantiated number is more than the one defined by policy P3.5, for the same QC and TT, or if the specified allocation exceeds the maximum allowed by policy P3.1.

Conflicts related to DRsM policies are mainly dynamic and arise due to ND policies constraining the run-time allocation of resources. These are inter-module conflicts and are a result of the hierarchical relationship between the two modules where policies introduced at the ND level are refined, communicated, and executed by DRsM as well. More specifically, an *ndMaxViolation* conflict occurs when policy P4.1 tries to increment the allocation for a QC but the calculated BW exceeds the upper bound set by the ND directive P3.1. Similarly, an *ndMinViolation* conflict occurs when policy P4.2 tries to decrement the allocation but the calculated BW is less than the lower bound set by the ND. Another high-level directive that is refined down to the DRsM level is a general resource management policy, which explicitly specifies that during a DRsM operational cycle, the full link capacity should be allocated between the various QCs. This implies that a DRsM policy action aiming to increase or decrease the allocation for a specific QC will violate the above rule as the resulting allocation may exceed or be less than the link capacity. We term these inconsistencies as *overAlloc* and *underAlloc* conflicts respectively. The last of DRsM related conflicts is an intra-module conflict and involves policies responsible for the computation of new thresholds and allocation of resources. The inconsistency arises if the allocated BW is below its respective upper utilization tracking threshold, in which case a threshold incompatibility (*thrshIncompat*) conflict should be signaled.

V. Conflict Analysis and Tool Support

The conflict analysis approach presented in this paper has two main aspects: the definition of appropriate rules for determining potential conflicts in policy specifications, and the effective deployment of analysis processes in the context of the managed environment. Detection rules are used to describe the conditions under which a conflict will arise and include information from policies and the managed environment itself to cater for the various QoS management conflicts. Although guidance can be provided by the policy refinement process for some conflict types – in the case of ME conflicts the actions associated with multiple disjunctive sub-goals should be encoded in a detection rule – their specification is a manual process largely relying on the knowledge of an expert administrator. Since most of the defined policies are generic to a certain extent, re-use of the conflict detection rules in other QoS frameworks could be possible. A comprehensive set of detection rules together with system-specific information is used by the analysis processes to determine potential inconsistencies.

The principal challenges in detecting policy conflicts are being able to account for the constraints that limit the applicability of a given policy to specific states of the managed system and the effects of enforcing policies on the states of the managed system. To achieve this, it is necessary to use formal reasoning techniques and formal models of the QoS management system behavior, policy enforcement mechanisms and the policy rules themselves. Since the QoS management system and the policy enforcement mechanisms concerned are event-based reactive systems, we have used Event Calculus as the underlying formal representation. In addition to having built-in representations for events and persistence of properties Event Calculus is a suitable formalism because it supports both deductive and abductive reasoning.

Fig. 5 outlines the architecture of our approach. Here, we distinguish between static and dynamic analyses as two different processes that are executed at different timescales and physical locations. Static analysis is an integral part of a Policy Management Tool (PMT) and is initiated by a network administrator, whereas dynamic analysis runs in a Policy Management Agent (PMA) and its execution is based on run-time events generated by the network/managed system. Both processes are semi-automated. In the case of static analysis a set of pre-specified rules are used to search the policy space for conflicts whose resolution is manual. Dynamic analysis also makes use of conflict rules, but the invocation of the process and the subsequent detection of conflicts are automated. The run-time resolution process is also automated and is based on a pre-defined set of policies. The details of the two processes are presented in the next sub-sections.

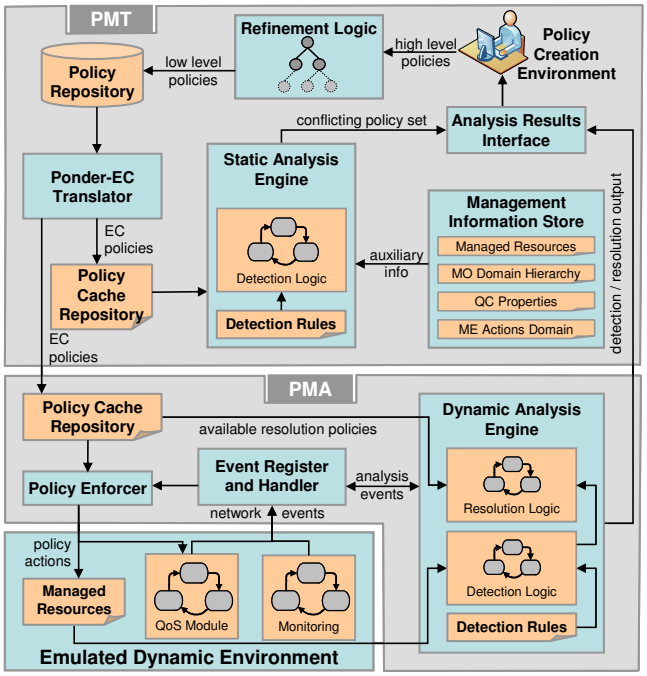

Fig. 5. Conflict analysis architecture. Static analysis is a centralized process running within the PMT, whereas dynamic analysis is a distributed process integrated within PMAs.

A. Static Analysis

As shown in the upper part of Fig. 5, our approach towards static conflict detection is based on the output of a refinement process [11, 17], where high-level policy specifications introduced by a network administrator are decomposed into low-level implementable ones and mapped onto their respective EC representation. Before their enforcement, policies are analyzed by the static analysis engine, a process initiated by a user administrator, by performing comparisons on a pair-wise basis. This process makes use of specialized detection rules, which are pre-specified by an expert and loaded to the engine, but also domain-specific information regarding the QoS management modules, to identify inconsistencies.

Based on the identified conflict types, we have defined a set of rules expressed in the form of logic predicates that encapsulate the conditions to be met for a conflict to occur. These predicates are used as conflict fluents in EC notation and can be considered as goal states that, when achieved, signify the detection of a conflict. The advantage of using such a methodology is that, in addition to detecting possible conflicts, an explanation as to why a conflict occurred will always be provided.

The conditions for a conflict can either be acquired from the policy specification itself but also from domain-specific information. The example predicate below is based solely on information provided by policies and aims at detecting redundancy conflicts by matching certain key parameters as well as actions in the policy specification through a pattern directed search. The variables of the *conflictData* term are unified during the search if the conditions for a conflict, as

defined in the predicate, hold, thus providing all the information pertinent to the conflict.

```
holdsAt(conflict(redundancy, conflictData(PolID1,
                 PolID2, QC1, QC2, OQL1, OQL2)), T) ←
     holdsAt(oblig(PolID1, Subj, op(Targ,
                   Action[Params1])), T) ∧
     holdsAt(oblig(PolID2, Subj, op(Targ,
                   Action[Params2])), T) ∧
     (numParams(PolID1, PolID2) == 1 ∨
      numParams(PolID1, PolID2) >= 2 ∧
      intersect(Params1, Params2, Params) ∧
      Params \== []) ∧
     PolID1 \== PolID2.
```

Apart from redundancy conflicts the detection of all other inconsistencies requires not only information provided by the policy specification but also QoS-specific information, such as properties of the managed resources and the supported QoS classes. These are encoded in the detection rules as further constraints that should not be violated. The example below represents the relevant predicate for detecting QoS priority conflicts among policies for setting the quality level of traffic classes. As described in Section IV-B, this conflict will be detected if the defined quality level of a QC with high priority is less than that of a QC with a lower priority.

```
holdsAt(conflict(qcPriority, conflictData(PolID1,
                 PolID2, QC1, QC2, OQL1, OQL2)), T) ←
     holdsAt(oblig(PolID1, Subj, op(Targ,
                   setQltLvl(QC1, OQL1))), T) ∧
     holdsAt(oblig(PolID2, Subj, op(Targ,
                   setQltLvl(QC2, OQL2))), T) ∧
     ((QC1::priority > QC2::priority ∧ OQL1 < OQL2) ∨
      (QC2::priority > QC1::priority ∧ OQL2 < OQL1)) ∧
     PolID1 \== PolID2.
```

The output of the static analysis process is a set of conflicting polices along with an explanation of their occurrence [5]. The resolution of these conflicts is a manual process that has to be carried out by the user administrator, in which policy parameters are modified or, in the case of redundancies, policies are eliminated from the system. A number of conflict resolution methods aiming to automate the process have been proposed in the literature, most of which are based on policy priorities [2, 18, 23, 34]. These allow two potentially inconsistent policies to coexist within the system and involve determining which of the two should prevail in the event of a conflict. This is possible due to the nature of the conflicts considered for which precedence between contradicting policies can be established on the basis of modality, specificity, or recency. The static conflicts identified in this work however, do not allow for automation in their resolution since they mostly occur due to inconsistent policy action parameters rather than just actions. Taking the example of *qcPriority* conflicts, consider two instances of policy P1.2 with which the OQL values associated with EF and AF1 traffic are set to 0.8 and 0.9 respectively. Resolution of this conflict can be achieved by either resetting the OQL value of AF1 traffic to be equal or less than that of EF traffic, or vice versa. Since there is no clear indication as to which of the two strategies to follow, and also because the amount by which the OQL will change, for either, QC has an impact on the overall QoS provisioning objectives, human intervention is unavoidable. The details associated with the inconsistency however, can guide the administrator when correcting it.

B. Dynamic Analysis and Conflict Resolution

While the analysis process described in the previous section is able to deal with static conflicts, some inconsistencies can only be detected at policy enforcement time as they depend on the current state of the network and the resulting configuration output of on-line QoS management modules (SLS-I and DRsM). For this reason, the process for handling dynamic conflicts needs to be embedded within a PMA which has access to the run-time information required.

Detection of dynamic inconsistencies is still based on a set of pre-specified conflict predicates, which, in this case, require additional information regarding the run-time state of online modules. In the context of our work, the conditions under which a conflict will arise are represented by constraints that depend on the conflict type. The rules for detecting such conflicts are based on the fact that two or more policies violate these constraints. The *conflict(srMaxViolation, …)* fluent below is such an example and indicates the violation of an SLS-S refined directive defining the maximum service rate for a TT. Here, the constraints conveyed to the conditional part of the predicate include the specific policy actions with matching TT parameters, and the actual value of *required* rate calculated by SLS-I. The latter is represented as an argument of the *reqSR* term and the conditions for a *srMaxViolation* conflict will be satisfied if this value exceeds the maximum rate specified by the SLS-S refined policy. Similar rules have been defined for all the identified dynamic conflicts relating to both SLS-I and DRsM policies [6].

```
holdsAt(conflict(srMaxViolation, conflictData(PolID1,
                 PolID2, TT, SR_FS, SR)), T) ←
     holdsAt(oblig(PolID1, Subj, op(Targ,
                   incrSR(TT, Value))), T) ∧
     holdsAt(oblig(PolID2, Subj, op(Targ,
                   setSR_FS(TT, SR_FS))), T) ∧
     reqSR(TT, SR) ∧ SR > SR_FS.
```

Despite the fact that the resolution of static conflicts is performed manually, this process takes place before policies are deployed in the system and does not impose any run-time overheads on the functionality of on-line modules. Dynamic conflicts however, require system components to both detect and resolve conflicts in real-time, without degrading the performance of the system. This has been the main motivation behind our dynamic analysis approach, which aims at automating the triggering of the detection process and the handling of conflicts at run-time.

Unlike other resolution methodologies [2, 18, 23], our approach does not involve identifying which of the conflicting policies will prevail based on their relative priority, but provides separate resolution rules that handle potential inconsistencies. These rules are effectively obligation policies which are pre-specified by the administrator using the Ponder format and their triggering events are conflict occurrences rather than network events. Although less automated than

precedence-based solutions where resolutions do not require to be defined prior to analysis, our approach is more flexible since: (a) it overcomes the problem where precedence cannot be established, and, (b) custom resolution rules can be provided for handling different conflicting situations that may arise. Once created, resolution policies are translated to their respective EC representations and communicated to the PMA, as shown on Fig. 5, where they are stored in a local cache repository. They are triggered by events generated by the dynamic analysis engine and their enforcement results in resetting the system into a state in which a conflict is resolved.

It is evident that dynamic conflicts arise as a result of a change in the state of a managed module, which in turn is caused by the enforcement of a new policy. To enable the automatic deployment of the analysis process, the latter needs to be notified of such events. This is achieved by processing the detection rules a priori and extracting information about policy actions that can potentially cause a conflict when enforced. These are encoded in the first field in the conditional part of detection predicates as in the *srMaxViolation* example, and are used to configure the *Event Handler* of Fig. 5. When the latter intercepts a policy enforcement event matching an action that can potentially lead to a conflict, the analysis process is notified and detection is initiated for that specific conflict. If the latter materializes, the resolution logic of Fig. 5 enters a *resolving* state which performs a search in the cache repository for a possible resolution pertaining to the detected conflict. If an appropriate resolution policy is identified a notification is passed back to the Event Handler which in turn triggers the enforcement of the resolution.

In contrast to the identified static inconsistencies, one of the two policies involved in a dynamic conflict acts as a constraint, the violation of which is the very reason for the occurrence of that conflict. The use of resolution policies is enabled by the fact that the constraining value can be considered as the "strong" value and is therefore used to quantify the relevant parameter in a resolving policy action. Extending the example of the *srMaxViolation* conflict, the resolution policy below sets the service rate to the maximum permissible value defined by the relevant SLS-S directive. The resolving value, $SR_{FS,}$ can be acquired from the parameters of the SLS-S policy on the fly – as this quantifies the relevant variable in the conflict predicate – and instantiate the associated parameter in the resolution policy action. The latter can be re-used for multiple occurrences of the same inconsistency alleviating the need for human intervention. Furthermore, since the resolution rules are part of the formal description, the analysis engine can determine which resolution policy applies for a particular conflict predicate based on the information provided for that conflict.

```
initiates(sysEvent(conflDetected(srMaxViolation,
          conflictData(PolID1, PolID2, TT, SRFS))),
          oblig(resPol1, slsiPMA, op(servAdjustMO,
                setSR(TT, SRFS))), T).
```

The work in [33] describes an alternative approach for the handling of dynamic inconsistencies and follows the validation principle of [32]. The authors propose the use of constraints to prevent a policy from firing if a new configuration parameter is not consistent with an associated system variable. Although this approach can prevent a run-time conflict, it may also prevent the system from making a potentially essential reconfiguration to meet an SLS requirement. Consider, for example, the *fully satisfied* service rate (SR_{FS}) of a trunk to be 100Mbps, and a policy that increases the rate allocated to that trunk by 20% when executed: if the policy triggering condition is met when the current allocation is at 90Mbps, the constraint will prevent the policy from firing and as a result the rate allocation will remain unchanged. Our approach overcomes this problem and allows for the correct configuration of resources, which, in this case, is the maximum permissible value of SR_{FS}. A practical example of the run-time analysis process is demonstrated in Section VI-B.

C. Conflict Analysis Tool

The proposed analysis techniques have been developed and integrated into a policy conflict analyzer. This tool implements static and dynamic analysis engines based on Prolog and its deductive reasoning capabilities, supports Ponder policy specifications and has mapping capabilities to formal representation, integrates emulated execution environments of online QoS modules, and provides a conflict analysis user interface (Fig. 6).

Static analysis implements the various conflict types identified in Section IV and takes as input Ponder policies that have been previously converted into the Event Calculus representation. The reasoning engines iterate through the policies to match the conditions specified in EC conflict rules, and outputs a set of conflicting policy pairs, along with an explanation of their occurrence. For demonstrating our dynamic analysis approach, the tool integrates a run-time execution environment that emulates the behavior of online modules through state machines. This is an Event Calculus based model of the system which allows the enforcement of policy actions and can also generate user-defined events about emerging network conditions. This is a temporary placeholder serving experimentation purposes; we plan to hook our dynamic analysis engine with the real execution environment in the future. Dynamic reasoning engines interface with the run-time environment through an event handler, which provides a two-way notification service, allowing for an efficient and automated run-time analysis process, including detection invocation, and conflict resolution.

The user interface consists of three main panels corresponding to static, dynamic analysis, and presentation of results. The first allows the user to initiate detection queries for static conflicts choosing among different inconsistencies to look for, whereas the dynamic panel allows the user to interact with the run-time environment by entering network events which can trigger the detection and resolution of potential conflicts. Lastly, the analysis results panel is shared by both

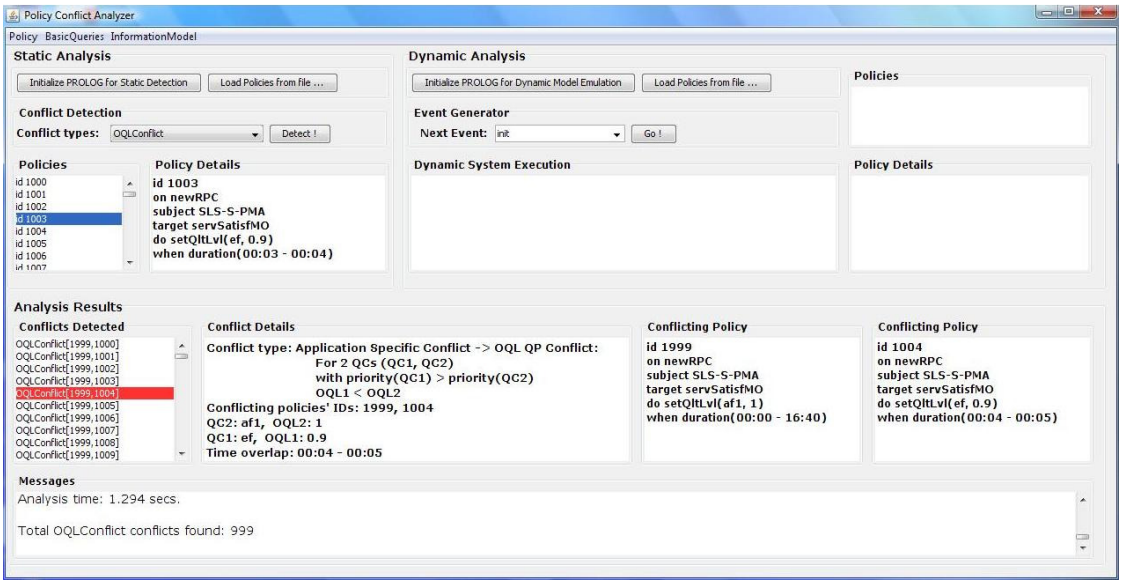

Fig. 6. Conflict analysis tool interface, with three main panels corresponding to static analysis, dynamic analysis and presentation of results. Here we show an example of analyzing 1000 policies for static conflicts and display the details of a QoS class priority conflict among SLS-S policies that set the quality level of EF and AF1 traffic classes.

processes and displays the output of conflict analysis by mapping the EC format in a user friendly representation. The example of Fig. 6 depicts the results of statically detecting *qcPriority* conflicts among SLS-S policies. A demonstration of the capabilities of the tool involving dynamic analysis is provided in the next section.

VI. Conflict Analysis Tool Evaluation

This section presents the results of a number of experiments conducted to evaluate the performance, scalability, and correct operation of the static and dynamic analysis engines developed. All experiments were performed on a Centrino Duo 2GHz processor with 2GB of RAM, and subject of the conflict analysis were service management policies.

A. Conflict Detection Performance Analysis

The main aim of the performance evaluation experiments concerning static conflict analysis is to determine the relative times taken to detect inconsistencies among varying numbers of policy specifications. Performance is primarily influenced by the evaluation of a conflict predicate in terms of: (a) the cost in evaluating its conditions, and (b) the number of times it is evaluated. Since experimentation showed that the number of conflicts only has a minor impact on performance, the experiments described below consider the number of policies, their type, and QoS-specific information as the main factors affecting the evaluation of conflict predicates.

Experiment 1

To investigate the impact of policy types in the analysis, a module-independent conflict is required which can detect the same inconsistency among different policy types and can ultimately provide a uniform basis upon which to compare performance. As such, this experiment concerns redundancy conflicts detected over different numbers of policies. In the first case only one policy type is used – for setting the quality level of a single QoS class. The number of conflicts, although not having a substantial impact on the performance, is kept constant as the number of policies is varied. The number of times the conflict predicate is evaluated is defined by the number of policies since the detection process iteratively compares each policy with the rest in the set. This can be quantified by equation (1) below, where L is the number of policy types, and N_l is the number of policies of a particular type.

$$\sum_{l=1}^{L} \frac{N_l - 1}{2} \times N_l \qquad (1)$$

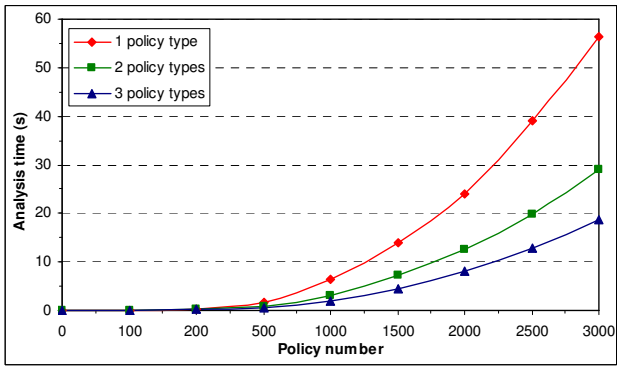

Fig. 7. Redundancy conflicts – detection performance against number of policies with varying policy types.

Fig. 7 demonstrates the performance of the detection process where the execution time grows quadratically with respect to the number of policies, namely $O(N^2)$. As suggested by (1), for 2500 policies of a single type the detection predicate is

evaluated 1999×10^3 times, which takes 39 seconds. Introducing more policy types, e.g. for setting service satisfaction factors and the upper limit in the RAB, the performance is significantly improved as the number of comparisons decreases, with all the conditions in the detection predicate only being fully evaluated when matching policy actions are found. For 2500 policies of two and three types, there is a performance improvement of 49% and 66% respectively. These results are validated against the theoretical gain provided by (1), which is 50% and 67%.

Experiment 2

Another factor that influences analysis performance is application-specific information. This is particularly important when dealing with QoS management conflicts whose occurrence depends on such information, as for example the number of QoS classes supported and their impact on determining *qcPriority* conflicts among SLS-S policies setting the service quality level. Equation (2) below can be used to calculate the number of times the relevant predicate is evaluated when detecting such conflicts, where M is the number of QCs involved, L is a counter equal to $M-1$, and N_l and N_m are the number of policies setting the quality level of particular QCs. For an example scenario involving three QCs, EF, AF1, and BE, policies setting the OQL of EF traffic are compared against the ones for AF1 and BE, and those for AF1 traffic against the ones for BE. It can be shown that $(N_1\times N_2)+(N_1\times N_3)+(N_2\times N_3)$ comparisons are performed, where N_1, N_2, and N_3 represent the number of policies associated with each QC.

$$\sum_{l=1}^{L}\sum_{m=l+1}^{M} N_l N_m \qquad (2)$$

Although of the same complexity of $O(N^2)$ as the previous experiment, the detection process for this conflict type is more expensive, as indicated by Fig. 8, especially with an increasing number of QCs. The experimental results indicate an increase of 35% in detection time between two and three QCs, and 53% between two and four, which are comparable to theoretical values of 33% and 50% obtained by (2) respectively.

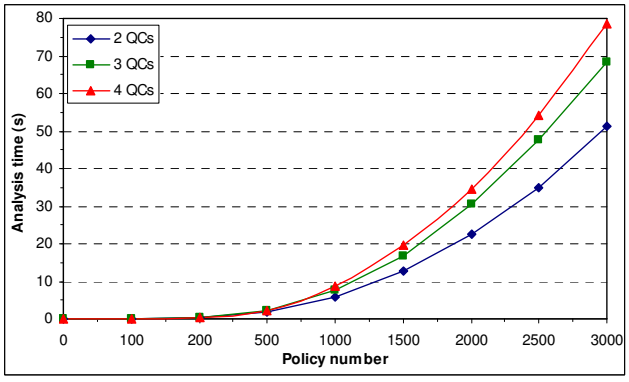

Fig. 8. Detection performance against number of policies with varying QCs.

Experiment 3

The last experiment compares the performance of various detection predicates. To provide a meaningful comparison this experiment involves a set of policies of the same type which is prone to more than one inconsistency. We consider *redundancy*/*qcPriority* conflicts among service quality policies for two QCs, and *redundancy*/*multiplexing* conflicts between almost and full satisfaction factor policies also for two QCs. In the first case the performance of the *qcPriority* predicate is substantially worse than that of *redundancy* by an average of 52% over a range of 3000 policies (Fig. 9), despite the fact that it is evaluated half as many times based on equations (1) and (2). This demonstrates the simplicity in detecting redundancies involving the matching of policy actions, and the cost associated with determining relative priorities between potentially conflicting QCs.

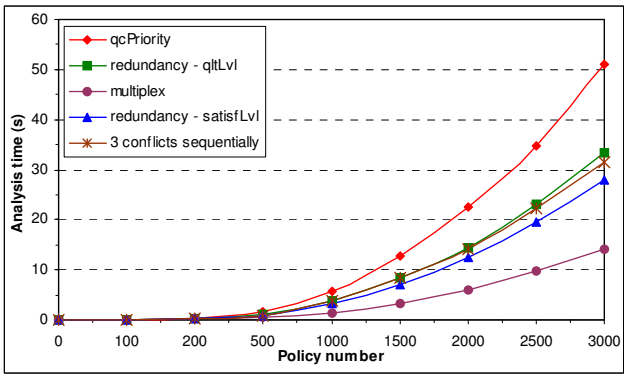

Fig. 9. Detection performance against number of policies for different conflict types.

The performance in detecting multiplexing conflicts is the most efficient with nearly 100% gain when compared to a redundancy analysis on the same set of policies. Equation (3) below can be used to calculate the number of multiplexing predicate evaluations, where L is the number of QCs, and N_{AS} and N_{FS} are the number of policies for almost and full satisfaction factors of a specific QC.

$$\sum_{l=1}^{L} N_{AS} N_{FS} \qquad (3)$$

For 2500 policies – $N_{AS} = N_{FS} = 625$ for each of the two QCs – equation (3) results in 781250 predicate evaluations, which is achieved in 9.7 seconds, whereas double the number of comparisons are required to determine redundancy conflicts in 19.6 seconds.

The last experimental result in Fig. 9 concerns the sequential execution of all three conflict rules, where half of the policy set consists of service quality policies and the other half is equally split between policies for almost and full satisfaction factors; 2 QCs are involved. The combined performance is better than two of the individual predicate evaluations, which is attributed to the policy set and the decreased number of predicate evaluations. The number of policies associated with the expensive *qcPriority* conflict, for

example, has halved resulting to a 75% decrease in evaluations of the relevant predicate.

B. Dynamic Analysis and Emulated System Behavior

As mentioned previously, the analysis tool developed interacts with the emulated dynamic behavior of online modules by enforcing policies to anticipate emerging conditions regarding the network status and eventually handle potential inconsistencies at run time. In contrast to static detection, dynamic analysis aims at discovering a single inconsistency at a time and enforcing the appropriate resolution policy. For this reason, searching the entire policy space for a conflict may not be required, thus improving the detection performance in comparison to static analysis. The evaluation of this approach, in addition to performance, is in terms of correct functional behavior in the event of a conflict.

To demonstrate the functionality of the dynamic analysis engine, we consider a scenario involving the SLS-I module, which is loaded with 100 policies, and concentrate on managing the service rate of a specific TT. The current allocation for this TT is 120Mbps with the almost and fully satisfied service rates set by SLS-S policies at 100Mbps and 150Mbps respectively. The screenshot on Fig. 10 shows the response of the analysis engine when an upper threshold crossing alarm is received.

Fig. 10. Analysis example of a dynamic conflict: detection and resolution.

The analysis process is initialized at T=0, at which point it goes through any loaded conflict specifications and extracts policy actions that can potentially cause a conflict when enforced. These actions are registered in the event handler which can in turn notify the detection engine upon events that activate such actions. In the example above, two such actions are extracted from conflict predicates relating to service rate violations. At T=1 the threshold alarm triggers a policy for decreasing the service rate by 25% down to 90Mbps, which activates the detection engine for a potential *srMinViolation* at T=2. The conflict materializes and the resolution engine is activated at T=3, which first determines and then triggers the appropriate resolution policy for this conflict type. At T=4 this policy is enforced configuring the service at the minimum acceptable (almost satisfied) rate. The cycle is completed with the analysis process going to idle state at T=5 consuming not more than 10ms. The delay introduced can be argued as being acceptable, even for the strict requirements of EF traffic, as long as conflicts do not occur extremely frequently. The lower part of Fig. 10 shows the specifics of the detected conflict including an explanation of the inconsistency, the policies involved, and the resolution enforced.

VII. Related Work

There has been considerable work on security policy conflict analysis over a number of years [20, 21, 26, 27] but interest in management policy conflict analysis is comparatively recent although increasing. The authors in [2] identify and classify a number of application-specific conflicts and describe the conditions under which they occur. They provide a methodology for their detection and resolution, the latter being the most popular approach in the literature, where policy precedence rules are used to define which of the conflicting policies is to prevail after a conflict has been detected. A similar approach is employed in [36] and [37] where different metrics have been used to establish precedence among policies including time, role, specificity, modality and numerical priority. The last method is also used in [19], which targets system management policies. This is part of a *ratification* process where new policies are approved before being committed in a system. The authors identify that meaningful numerical priorities are notoriously difficult for users to assign and may result in arbitrary priorities which do not really reflect the importance of policies. For this reason, they developed algorithms to automatically assign the priority values to new policies and to adjust the values of related policies when given only the relative priority of a new policy. The algorithms implement the conflict resolution module of the IBM's PMAC platform by maintaining ordered lists under policy insertion and deletion operations. Although resolution based on precedence may be useful in some occasions, we believe that this may not be a flexible solution to the problem, especially when application-specific environments are concerned, as demonstrated in some of our examples where new policies need to be enforced.

There are few conflict analysis examples that target specific application domains. The authors of [4], [20], [21], and [22]

have focused on techniques for analyzing legacy firewall policies for networks with centralized and distributed firewalls. All possible firewall rule relations are formally defined and are used to identify and classify policy conflicts (anomalies). Their resolution is based on the relative ordering of rules in a filtering policy and a degree of automation is proposed for some conflict types by removing or re-ordering rules. The main shortcomings of these approaches is the dependence on low-level legacy firewall policies to perform anomaly detection, the lack of explanations as to how and why conflicts occur, and failure to address possible inconsistencies that may arise at run-time.

Another application domain for which conflict analysis has been addressed is that of telecommunications and more specifically call control. In [29] the authors identify the analogy between policy conflicts and feature interactions, and provide a taxonomy of conflicts that is based on five principle dimensions. Depending on the manifestation of the various conflicts, they identify different approaches that could potentially be used to detect and resolve conflicts, which are based on techniques previously applied on feature interactions. This work is extended in [30] where specific resolution processes are proposed to handle call control policy conflicts both in centralized and distributed settings. The methodology is similar to the one presented in this paper and is based on the notion of resolution policies. The detection of conflicts however is not supported by a separate process, but the various conditions are encoded within resolution policies instead. Resolution specifications can thus become complex and their evaluation quite expensive. This work lacks assessment data which could evaluate the performance of the proposed approach.

Recent work in [35] targets the same application domain as the one considered in this paper. Here, the authors identify conflicts among policies managing QoS in DiffServ networks, but only tackle a small portion of the problem regarding resource management at the router level. The policies involved in this process define the treatment of a traffic flow on network nodes by setting parameter values for BW allocation, queue size, drop method, and priority for the various Per-Hop Behaviors (PHBs). Inconsistencies among these policies are classified according to the scope in which they occur: *intra-PHB* conflicts arise within the flow properties at a specific node and *inter-PHB* conflicts occur between policy definitions across different nodes.

Motivated by the advantages provided by information models in representing managed entities, such as platform and protocol independency, the authors in [38] propose their use in the process of conflict detection. More specifically, this work is based on the DEN-ng model [40], which, apart from managed entities, is also used for representing both the policies and the conditions under which these may conflict. This work has recently been extended in [39] to support the overall methodology and implementation of the conflict detection approach. Here, the authors describe a two-phase analysis algorithm, which when querying an information model, firstly determines the relationships between a pair of policies and secondly, applies conflict patterns to determine if the policies should be flagged as conflicting. Policy relationships are expressed in terms of policy subjects, targets and actions, while conflict patterns concern constraints defined in the information model describing policy relationships that must hold for a conflict. Determining a conflict involves transforming the above information into matrices and performing comparisons. The use of information models is also proposed by Kempter in [41], where invariants extracted from the models are used as indicators for conflicts when they are breached. Although this approach benefits from the inherent advantages of information models, XML representation of policies and conflicting conditions can become very verbose thus posing a cumbersome task for a network administrator if a manual change is required. Furthermore, the use of matrices in [39] limits the definition of relationships to the core fields of a policy. As such, conflicts that arise as a result of inconsistent action parameters, rather than actions, are difficult to detect and the exact reason for their occurrence cannot be provided.

Among the many alternative approaches to policy specification, there are a number of proposals for formal, logic-based notations. In particular, logic-based languages have proved attractive for the specification of security policy, as they support a well-understood formalism, amenable to analysis. However, they can be difficult to use and are not always directly translatable into efficient implementation. One such example is the Policy Description Language (PDL) [28], which is used for the specification of obligation policies. The language can be described as a real-time specialized production rule system to define policies. Later work by Chomicki [31], extends PDL to include the concept of action constrains, which are policies that prevent a specified action from being performed in a given situation. This work introduces the idea of using a policy monitor to detect conflict situations and resolve them by either suppressing the events that could lead to a conflict or overriding the conflicting action. Another work that proposes the use of logic-based specification of policies is [27]. The language proposed has relatively simple, well-understood semantics and policies are analyzed using deductive reasoning techniques. Resolution of potential conflicts among authorization policies relies on the use of precedence rules.

Work on computational efficiency for conflict detection and resolution mechanisms was presented in [23] and [24]. The authors identified several conflicts that may occur in open distributed systems and classified them into static and dynamic. Their detection mechanism involves identifying and predicting all possible conflicts at compile-time, based on knowledge of the temporal characteristics of the policies in the specification. In the case of dynamic conflicts the relevant conditions are stored in a database and subsequent monitoring of system events can lead to determining the occurrence of a conflict. Furthermore, they developed an approach as to when it is appropriate to resolve conflicts. Based on the fact that a

resolution process can be computationally intensive, they proposed different approaches according to the likelihood of a conflict occurring and the cost of resolving that conflict. The actual resolution methodology presented by the authors follows the guidelines provided in [2], where policy precedence rules are being used.

VIII. Conclusions and Future Work

This paper presented our approach towards policy conflict analysis based on the formalization and reasoning provided by Event Calculus and its application in the domain of DiffServ QoS management. The subject of the analysis techniques presented here is a set of management policies that can be used to influence/control the behavior of key modules in the process of QoS provisioning. The various inconsistencies that can arise between these policies have been identified and classified based on their characteristics, which are used to describe the reasons and the conditions under which a conflict will arise.

We define conflicts that can occur between policies applied to a single management module (intra-module), or between policies specified for different modules (inter-module) as a result of their hierarchical relationship, but the main characteristic distinguishing between conflicts is the time-frame at which they can be detected. This has driven the design and specification of two different methods to address the issues associated with the analysis of conflicts that can be detected statically, at policy specification-time, and those that can only be determined dynamically, during system execution, based on feedback regarding the current state of the managed system. These techniques have been implemented and integrated in a conflict analysis tool aiming to provide a network administrator with a usable interface through which to interact with the management system and perform both static and dynamic consistency checks.

Our implementation is heavily based on the use of Event Calculus for which we provide seamless and efficient mapping mechanisms used by the analysis engines. Its use allows the use of advanced reasoning methods and provides the means to not only identify a conflict but also provide an explanation as to how that conflict occurred. This is particularly important when guiding a network administrator to handle inconsistencies requiring manual resolution, as in the case of the static conflicts identified. To satisfy the requirements of dynamic conflicts with respect to efficiency, we concentrated in providing an automated run-time analysis process. This can be automatically invoked based on run-time network events, can provide a resolution if a conflict materializes, and also instruct the appropriate entity for the enforcement of that resolution. The latter is in the form of pre-defined policies that are generic enough with only few required per conflict type to cater for multiple occurrences of the same inconsistency. Finally, the tool developed has been used to perform extensive experiments through which it was possible to identify the main reasons that influence the performance of the analysis engines.

The future directions of this work are in the domain of collaborative QoS management, where neighboring network providers set-up service-level agreements aiming to create an end-to-end chain for the delivery of QoS sensitive applications. We envisage the negotiation process to be one where each provider tries to force its own policies in terms of requirements and objectives resulting in conflicting situations. A collaborative negotiating process would act as a mediator where an optimal solution, satisfying both entities, would be achieved through conflict analysis.

References

[1] J.D. Moffett, M.S. Sloman, “Policy conflict analysis in distributed system management,” Journal of Organisational Computing, vol. 4, pp. 1-22, 1994.

[2] E.C. Lupu, M.S. Sloman, “Conflicts in policy-based distributed systems management,” IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management, vol. 25, pp. 852-869, 1999.

[3] A.K. Bandara, E.C. Lupu, A. Russo, “Using Event Calculus to formalise policy specification and analysis,” proceedings of 4th IEEE Workshop on Policies for Networks and Distributed Systems, Lake Como, Italy, 2003.

[4] E. Al-Shaer, H. Hamed, “Modeling and management of firewall policies,” IEEE Transactions on Network and Service Management, Volume 1-1, April 2004.

[5] M. Charalambides, et al., “Policy conflict analysis for quality of service management,” proceedings of 6th IEEE Workshop on Policies for Networks and Distributed Systems, Stockholm, Sweden, 2005.

[6] M. Charalambides, et al., “Dynamic policy analysis and conflict resolution for DiffServ Quality of Service Management,” proceedings of IEEE/IFIP Network Operations and Management Symposium, Vancouver, Canada, 2006.

[7] N. Damianou, N. Dulay, E.C. Lupu, M.S. Sloman, “The Ponder policy specification language,” proceedings of 2nd IEEE Workshop on Policies for Networks and Distributed Systems, Bristol, UK, 2001.

[8] P. Trimintzios, et al., “Service-driven traffic engineering for intra-domain Quality of Service management,” IEEE Network Magazine Special Issue on Network Management of Multiservice, Multimedia, IP-based Networks 17(3): 29-36, 2003.

[9] M. P. Howarth, et al., “Provisioning for Inter-domain Quality of Service: the MESCAL Approach,” IEEE Communications Magazine, 2005.

[10] E. Borcocci, et al., “Admission control algorithm for aggregated pipes service invocation in multi-domain IP environment,” proceedings of 3rd International Workshop on Next Generation Networking Middleware, Coimbra, Portugal, 2006.

[11] A. Bandara, et al., “Policy refinement for IP differentiated services quality of service management,” IEEE Transactions on Network and Service Management, 3(2), 2006.

[12] R.A. Kowalski, M.J. Sergot, “A logic-based calculus of events,” New Generation Computing, vol. 4, pp. 67-95, 1986.

[13] A. Russo, R. Miller, B. Nuseibeh, J. Kramer, “An abductive approach for analysing event-based requirements specifications,” proceedings of 18th International Conference on Logic Programming (ICLP), Copenhagen, Denmark, 2002.

[14] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov. “Complexity and expressive power of logic programming,” proceedings of 12th Annual IEEE Conference on Computational Complexity (CCC'97), Ulm, Germany, 1997.

[15] P. Flegkas, P. Trimintzios, G. Pavlou, “A policy-based quality of service management architecture for IP DiffServ networks,” IEEE Network Magazine Special Issue on Policy Based Networking, vol. 16 No. 2, pp. 50-56, 2002.

[16] E. Mykoniati, et al., “Admission control for providing QoS in IP DiffServ networks: the TEQUILA approach,” IEEE Communications Magazine 41(1), 2003.

[17] J. Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, "A methodological approach toward the refinement problem in policy-based management systems," IEEE Communications Magazine, Topics in Network and Service Management, Vol. 44, No. 10, 2006.

[18] D. Agrawal, J. Giles, K.W. Lee, J. Lobo, "Policy ratification," proceedings of 6th IEEE Workshop on Policies for Networks and Distributed Systems, Stockholm, Sweden, 2005.

[19] D. Agrawal, J. Giles, K.W. Lee, J. Lobo, "Policy-based management of networked computing systems," IEEE Communications Magazine, vol. 43 No. 10, pp. 69-75, 2005.

[20] E. Al-Shaer, H. Hamed, "Discovery of policy anomalies in distributed firewalls," proceedings of 23rd IEEE Communications Society Conference (INFOCOM), Hong Kong, March 2004.

[21] L. Yuan, et al., "FIREMAN: a toolkit for firewall modeling and analysis," proceedings of IEEE Symposium on Security and Privacy, Oakland, CA, 2006.

[22] S. Ferraresi, S. Pesic, L. Trazza, A. Baiocchi, "Automatic conflict analysis and resolution of traffic filtering policy for firewall and security gateway," proceedings of International Conference on Communications, Glasgow, Scotland, 2007.

[23] Dunlop, J. Indulska, K. Raymond, "Methods for conflict resolution in policy-based management systems," proceedings of 7th International Conference on Enterprise Distributed Object Computing, Brisbane, Australia, 2003.

[24] N. Dunlop, J. Indulska, K. Raymond, "Dynamic conflict detection in policy-based management systems," proceedings of 6th International Conference on Enterprise Distributed Object Computing, Lausanne, Switzerland, 2002.

[25] P. Trimintzios, et al., "Quality of service provisioning through traffic engineering with applicability to IP-based production networks," Computer Communications, special issue on Performance Evaluation of IP Networks and Services, Vol. 26, No. 8, pp. 845-860, Elsevier Science Publishers, May 2003.

[26] V.D. Gligor, S.I. Gavrila, D. Ferraiolo, "On the formal definition of separation-of-duty policies and their composition," proceedings of IEEE Symposium on Security and Privacy, Oakland, CA, IEEE, May 1998.

[27] S. Jajodia, et al., "Flexible support for multiple access control policies," ACM Transactions on Database Systems 26(2), pp. 214-260, 2000.

[28] J. Lobo, R. Bhatia, S. Naqvi, "A Policy Description Language," proceedings of 16th National Conference on Artificial Intelligence, Orlando, Florida, USA, 1999.

[29] S. Reiff-Margainiec, K. J. Turner, "Feature interaction in policies," Computer Networks, 45(5), pp. 569-584, August 2004.

[30] L. Blair, K. Turner, "Handling policy conflicts in call control," proceedings of 8th International Conference on Feature Interaction, pp. 39-57, IOS Press, Amsterdam, June 2005.

[31] J. Chomicki, et al., "A logic programming approach to conflict resolution in policy management," proceedings of 7th International Conference on Principles of Knowledge Representation and Reasoning, Breckenridge, Colorado, USA, Morgan Kaufmann, April 2000.

[32] L. Lymberopoulos, E.C. Lupu, M.S. Sloman, "PONDER Policy implementation and validation in a CIM and differentiated services framework," proceedings of 9th IEEE/IFIP Network Operations and Management Symposium, Seoul, Korea, May 2004.

[33] R. Chadha, L. Kant, "Policy-driven mobile ad hoc network management," John Wiley & Sons, ISBN 978-0-470-05537-3, pp. 113-129, 2008.

[34] G. Russello, C. Dong, N. Dulay, "Authorization and conflict resolution for hierarchical domains," proceedings of 8th IEEE Workshop on Policies for Networks and Distributed Systems, Bologna, Italy, June 2007.

[35] T. Samak, E. Al-Shaer, H. Li, "QoS policy modeling and conflict analysis," proceedings of 9th IEEE Workshop on Policies for Networks and Distributed Systems, New York, USA, June 2008.

[36] J.Brashaw, et al., "Representation and reasoning for DAML-based policy and domain services in KAoS and nomads," proceedings of International Conference on Autonomous Agents and Multi-Agent Systems, Melbourne, Australia, July 2003.

[37] A. Uszok, et al., "New developments in ontology-based policy management: increasing the practicality and comprehensiveness of KAoS," proceedings of 8th IEEE Workshop on Policies for Networks and Distributed Systems, Bologna, Italy, June 2007.

[38] S. Davy, B. Jennings, J. Strassner, "Conflict prevention via model-driven policy refinement," proceedings of IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Dublin, Ireland, October 2006.

[39] S. Davy, B. Jennings, J. Strassner, "Application domain independent policy conflict analysis using information models," proceedings of IEEE/IFIP Network Operations and Management Symposium, Bahia, Brazil, April 2008.

[40] J. Strassner, "DEN-ng: achieving business-driven network management," proceedings of IEEE/IFIP Network Operations and Management Symposium, Colorado, USA, April 2002.

[41] B. Kempter, A.V. Danciu, "Generic policy conflict handling using apriori models," proceedings of IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Barcelona, Spain, October 2005.